VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

RayTracing na GPU

RayTracing on GPU

2014 Jan Nekvapil

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Nekvapil**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Raytracing na GPU
Raytracing on GPU

Zásady pro vypracování:

Cílem této bakalářské práce je implementace paralelního zpracování raytracing algoritmu s využitím GPU. Výpočet zobrazované scény je prováděn na grafické kartě pomocí platformy CUDA. Zobrazovaná scéna se skládá z bodových světel a objektů různého druhu povrchu ovlivňující optické vlastnosti objektu (lom, odraz či rozptyl paprsku). Urychlení výpočtu vykreslování scény kromě samotné paralelizace algoritmu bude zajišťovat dělení scény KD stromem a rovněž použití adaptivního antialiasingu a adaptivní kontroly hloubky. Výstupem práce bude také měření časové efektivity algoritmu v závislosti na náročnosti vykreslované scény.

Seznam doporučené odborné literatury:

[1] S. Guntury, P. J. Narayanan, "Raytracing Dynamic Scenes on the GPU Using Grids," IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 1, pp. 5-16, January, 2012
[2] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. 2007. Interactive k-d tree GPU raytracing. In Proceedings of the 2007 symposium on Interactive 3D graphics and games (I3D '07)
[3] Shevtsov, M., Soupikov, A. and Kapustin, A. (2007), Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. Computer Graphics Forum, 26: 395–404
[4] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In Proceedings of Graphics Hardware (2006), pp. 67–77.
[5] Hapala, M. and Havran, V. (2011), Review: Kd-tree Traversal Algorithms for Ray Tracing. Computer Graphics Forum, 30: 199–213

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lukáš Zaorálek**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 .....

Rád bych zde poděkoval svému vedoucímu práce, Ing. Lukáši Zaorálkovi, za vedení a rady, a své partnerce a rodině za podporu. Bez nich by tato práce nevznikla.

Abstrakt

Problém s vykreslením 3D scény na obrazovku je zde od počátků počitačové grafiky a k jeho řešení vede řada cest. Jednou z nich je raytracing, který dokáže vhodně pracovat s matematicky popsanými tělesy ve scéně a přirozeně vytváří řadu optických efektů, které musí jiné techniky simulovat.

Tato práce řeší základní implementaci raytracingového enginu na platformě CUDA a snaží se pomocí jednoduchých prostředků, jako je adaptivní hloubka či KD strom, dále urychlit zpracování scény.

Klíčová slova: Raytracing, CUDA, KD Strom, Antialiasing, Adaptivní hloubka, Adaptivní Antialiasing

Abstract

From the beggining of computere graphics there was a struggle to render 3D scene onto monitor screen and it was solved in various ways. One of them is raytracing, which can easily draw mathematicaly described objects in scene and naturaly creates variety of optical effects that have to be simulated by other techniques.

In this thesis, I present basics implementation of raytracing engine on CUDA platform and follow with simple ways to additionaly improve performance such as adaptive depth or KD tree.

Keywords: Raytracing, CUDA, KD Tree, Antialiasing, Adaptive Depth, Adaptive Antialiasing

Seznam použitých zkratek a symbolů

GPU	–	Graphic Processing Unit
CPU	–	Central Processing Unit
CUDA	–	Compute Unified Device Architecture
OpenCL	–	Open Computing Language
SIMD	–	Single Instruction Multiple Data
OOP	–	Objektově Orientované Programování
OpenGL	–	Open Graphics Library
GLEW	–	OpenGL Extension Wrangler Library
freeGLUT	–	free OpenGL Utility Toolkit
GLEW	–	OpenGL Extension Wrangler Library
SAH	–	Surface Area Heuristic
KD strom	–	K-dimenzionální strom
BVH	–	Bounding Volume Hierarchy

Obsah

1 Úvod

Člověk se odedávna snažil nějakým způsobem zachytit svět kolem sebe, či své představy a nějakým způsobem je zachovat, či předat ostatním. Lidé měli stále chuť zvěčnit svět tak, jak jej viděli všemi dostupnými prostředky a tak dali vzniku široké škále barev,dále pak následovala fotografie a film.

S příchodem počítačových terminálů začalo být vynakládáno velké úsilí do co nejdůvěrnějšího promítnutí reality na obrazovku. Z počátku to byla často pouze značná abstrakce v podobě textové ascii grafiky a podobnosti obrysu. Později se začaly objevovat barvy a snaha o důvěryhodné promítnutí objektu v prostoru, což vzhledem k dvou rozměrnosti obrazovek a omezené škále výstupních barev nebylo vůbec snadné. Definovat scénu v prostředí počítače není triviální záležitost. Musíme pečlivě zvolit nezbytná data a generalizovat kde to jde. Zde již přichází na řadu moderní renderovací techniky, které za použití různých druhů informací a jejich zpracování, dosahují každá různých druhů úspěchů v této oblasti.

Jednou z těchto technik je právě raytracing, který svou podstatou přirozeně naplňuje jeden z hlavních činitelů realističnosti vykreslené scény. Tím je její osvětlení, tedy kombinace světla a stínu a jejich působení na tělesech. Jak již název techniky napovídá, jejím jádrem je sledování (trace) paprsku (ray), konkrétně světelného. Ačkoli bychom podle této nápovědy chtěli sledovat foton od zdroje světla, tak jak letí prostorem a odráží se od objektů, mění směr, barvu a další vlastnosti, až v nějakou chvíli dorazí k našemu oku a vytvoří tak část obrazu, byla by to však velká ztráta času, protože drtivá většina těchto světelných fotonů by buď vyletěla pryč ze scény, nebo vlivem útlumů při odrazech skončila na některém z těles mimo pozorovatelnou plochu. Proto když mluvíme o raytracingu, mluvíme vlastně o "reverzním" raytracingu, kdy vypouštíme paprsky světla z oka kamery směrem do scény skrz promítací plochu (obrazovku) a sledujeme od kterých těles se odrazí a jestli v daných bodech je světlo či stín.

Cílem této práce je implementace raytracing enginu, který bude pracovat s odrazem a bodovými světly. Implementován bude paralelně na platformě CUDA. Při hledání průsečíků bude využívat k urychlení strukturu KD stromu předem vytvořeného na CPU. Dále bude pro urychlení používat adaptivní hloubku a pro vyšší kvalitu výsledného obrazu, opět s přihlédnutím k výkonu, adaptivní antialiasing. Celá implementace raytracingu by měla klást důraz na rychlost vykreslování statické scény.

2 Algoritmus

V úvodu jsem již popsal základní myšlenku algoritmu raytracingu, a to sledování paprsku světla z pozice pozorovatele směrem k objektům ve scéně.

Jednoduchý slovní popis algoritmu by tedy zněl následovně:
Veď paprsek z oka kamery skrz každý pixel a zjisti nejbližší průsečík s objekty ve scéně, vrať barvu tohoto tělesa.

Tento algoritmus ovšem nepočítá se světlem, odrazy, či lomy paprsků. Počítá pouze s tím, čemu říkáme primární paprsky, které nám samy o sobě příliš důvěrné informace o scéně neposkytnou. Vše co nám sdělí je, které těleso je skrz daný pixel vidět a jaká je jeho barva pokud bereme v potaz ambientní osvětlení. Proto bychom tento algoritmus měli doplnit o test, jestli je z nalezeného průsečíku vidět nějaké světlo tím, že ke každému vyšleme vlastní paprsek a opět otestujeme na průsečík se všemi objekty abychom zjistili zda je ve stínu. Dále můžeme spočítat odraz paprsku a sledovat tento odražený paprsek na jeho cestě a opakovat celý proces se světly a odrazem stále dokola, dokud nezůstaneme bez dalšího průsečíku s tělesem, nebo neukončí cestu paprsku nějaký další parametr. Takový algoritmus pak můžeme vidět ve výpisu 1

```
TraceRay(ray, depth)
{
    if (depth > maximal depth)
        return 0;
    intersection = closest intersection with objects in scene;
    if ( intersection exists )
    {
        for each ligt in scene
            color += light contribution ;

        if (surface is reflective )
            color += TraceRay(reflected ray, depth +1);

        return color modified by object properties ;
    }
}

for each pixel
{
    compute ray from camera through pixel;
    pixel color = TraceRay(ray, 0);
}
```

Výpis 1: Algoritmus Raytracingu

Pokud se na algoritmus podíváme v tuto chvíli zjistíme, že již tak triviální není a navíc v něm ve scéně s rozumným počtem objektů velmi rychle přibývá počet paprsků, ke kterým hledáme průsečík se všemi tělesy ve scéně. Proto se v raytracingu zavádějí urychlovací struktury, což jsou struktury udržující seznam objektů a další informace v takovém uspořádání, aby mám umožnily co nejmenším množstvím testů vyřadit co největší množství objektů z kandidátů na průsečík s paprskem.

Náročnost algoritmu ale i při použití urychlovacích struktur zůstává velmi vysoká, a tak jeho praktické využití v real-time grafice velmi omezené. Naštěstí není příliš náročné si povšimnout, že i přes výslednou složitost a náročnost algoritmu zůstávají jednotlivé paprsky stále navzájem zcela nezávislé a otevírají se tak možnosti paralelizaci a to masivní, zvláště pokud se nebudeme dívat na „několikajádrová“ CPU ale na mnoha desítek- až set-jádrová GPU, která jsou již v dnešní době běžně dostupná a to včetně programovacího rozhraní jako je CUDA nebo OpenCL.

3 Historie

3.1 Před dobou počítačů

Princip raytracingu byl využíván již velmi dlouho před vymyšlením prvního počítače a vznikem počítačové grafiky. Za první využití tohoto principu by se dal považovat nástroj používaný německým renesančním malířem Albrechtem Dürerem. Tento princip byl také u vzniku optické čočky a Sir Isaac Newton pojednává ve své knize o zákonech lomu a odrazu, což jsou již počátky vnímání sekundárních paprsků.

3.2 Počátky v počítačové grafice

V počítačové grafice se poprvé algoritmus objevil jako „ray casting“ popsaný Arthurem Appelem v roce 1968 [1]. Tento algoritmus počítal pouze s primárními paprsky a stínování objektu bylo poté prováděno tradičními metodami. Rekurzivní raytracing se sekundárními paprsky poté navrhl Turner Whitter v roce 1979 [2].

3.3 Paralelní zpracování

Dalším milníkem v historii raytracingu byl nástup více jádrových CPU. Paralelizace algoritmu začala otevírat dveře použití této techniky v real-time prostředí. Vznikla série projektů převádějící hry Quaqe do raytracingu [3], do kterých se následně zapojil i Intel jakožto největší výrobce CPU na trhu a vydal vlastní soubor raytracing kernelů Embree, které se snaží maximálně využít výhody CPU jako například SIMD (Single Instruction Multiple Data)

Zatím poslední a největší šancí pro real-time raytracing bylo zpřístupnění výpočetní síly GPU skrze platformu CUDA vyvinutou společností Nvidia pro jejich grafické karty. Velké množství jader GPU umožňuje masivní paralelizaci a i přes mnohem nižší výpočetní výkon jednoho jádra GPU oproti CPU a další omezení, dosahuje raytracing na GPU slibných výsledků. Nvidia vyvinula vlastní ray tracing engin OptiX založený na platformě CUDA.

4 Další metody renderingu

4.1 Rasterizace

Nejrozšířenější metoda renderingu, oblíbená pro svoji rychlost, je založená na prostém promítání 3D polygonů (trojúhelníků) do 2D průmětny monitoru. Nevýhodou tohoto řešení oproti raytracingu je, že neposkytuje žádné přirozené vizuální efekty a tak světlo, stín, odraz a další musí být řešeny samostatně, často oklikou v shaderech. Dalším mínusem je pak vykreslování pouze v polygonech, což způsobuje, že kvalita zobrazení oblých tvarů, jako jsou koule či kužely je závislá na počtu polygonů, který negativně ovlivňuje rychlost a další zpracování v shaderech.

4.2 Ray casting

Poprvé formulován v roce 1968 [1], jak bylo zmíněno již v úvodu jedná se de facto o raytracing počítající pouze s primárními paprsky. Jako takový získává výhodu snadného vykreslování matematicky popsatelných těles, ale jejich stínování již musí řešit sám.

4.3 Radiosity

Tato metoda také pracuje s paprsky světla, ale na rozdíl od raytracingu je nesleduje z oka pozorovatele, ale přímo mezi objekty ve scéně. Objekty jsou rozděleny na menší „plošky“ u kterých je pak vždy v páru porovnáván vzájemný vliv. Většinou se zpracovává v krocích, které sledují postupné šíření světla. Po prvním kroku jsou viditelné pouze plošky, které byly vystaveny přímému osvětlení, v dalších jsou pak osvětlovány ty, které jsou ovlivněny ploškami z prvního kroku. Jednotlivé objekty zde mohou tedy fungovat jako zdroje světla a výsledkem jsou měkké stíny a vzájemné ovlivňování barev objektů. Další výhodou je, že takto vyhodnocená scéna není závislá na pozici pozorovatele, takže jednou zpracovaná scéna je poté pozorovatelná z jakéhokoliv místa. Díky této vlastnosti a stupni realističnosti, kterou radiosita poskytuje, nachází využití ve vizualizaci interiérů, kdy je po delší úvodní inicializaci možný plynulý „průlet“ scénou. Nevýhodou je práce pouze s difúzním světlem, spekulární složka chybí právě kvůli nezávislosti algoritmu na pozici kamery. Časová náročnost je ještě vyšší než u raytracingu, i když paralelizace zde také znamená výrazné zlepšení. Další informace můžete najít v článku [4]

4.4 Path tracing

Je rendering založený na metodě Monte Carlo [5], tedy náhodném vzorkování. Paprsky jsou vystřelovány ze světla směrem do scény a po náhodném počtu odrazů je výsledný vzorek promítnut na obrazovku. Stejně tak můžeme pro vzorek sledovat paprsek z bodu na objektu, dokud nenarazí na zdroj světla. Využití obou těchto postupů najednou se říká obousměrný path tracing. Všechny tyto výsledky jsou poté průměrovány a tvoří výslednou barvu pixelu. K vytvoření obrazu je tedy zapotřebí velkého počtu vzorků, protože mnoho paprsků z plochy nikdy na světlo nenarazí a stejně tak paprsky ze světla často nedopadnou na viditelnou plochu, což se projevuje nepříjemnou zrnitostí obrazu při nízkém počtu vzorků na pixel. Path tracing je ze všech zmíněných metod nejnáročnější, ale při dostatečném počtu vzorků také nejvěrnější realitě a používá se jako reference pro ostatní metody.

5 CUDA

CUDA (Compute Unified Device Architecture) je platforma zpřístupňující výpočetní sílu GPU pro paralelní výpočty vyvinutá společností NVIDIA pro jimi vyráběné grafické karty. Platforma jako taková se skládá ze dvou částí a to hardwarové a softwarové architektury pro GPU. Na hardwarové úrovni je CUDA podporovaná nejznámější architekturou Fermi a na softwarové pak poskytuje rozhraní v podobě CUDA C/C++ postaveného nad jazykem C a CUDA Fortran, spravované přímo společností NVIDIA. Softwarové rozhraní CUDA poskytuje přímý přístup ke speciálním instrukcím a paměti GPU podporujících tuto platformu a umožňuje tak využití GPU k všeobecně výpočetně intenzivním úlohám, ne pouze k počítání a vykreslování počítačové grafiky, což bylo do té doby majoritním využitím GPU. Kromě NVIDIí spravovaných CUDA C/C++ a CUDA Fortran architektura podporuje také další rozhraní jako OpenCL od Khronos group (jíž je NVIDIA součástí) a DirectCompute od Microsoftu. Knihovny třetích stran, poskytující přístup k platformě, jsou dostupné také pro širokou škálu dalších jazyků, jako například oblíbená Java, Python či přímo pro matematické výpočty určené prostředí a jazyk MATLAB.

5.1 Historie

První verze CUDA SDK byla zpřístupněna veřejnosti 15. února 2007 a podporovala operační systémy Windows a Linux. Později byla přidána podpora Mac OS a dnes již běží na většině rozšířených operačních systémech. První série grafických karet podporujících platformu CUDA byla série G8x a od té doby ji podporují všechny grafické karty NVIDIA, včetně herně zaměřené série GeForce. Zatím poslední vydaná verze nese číslo 5.5. a byla uveřejněna 2. Srpna 2013. Dostupná je také již testovací verze 6.0 přinášející novinky v oblasti sdílení paměti mezi CPU a GPU bez nutnosti ji explicitně kopírovat, jak tomu bylo doposud.

5.2 Využití

CUDA odemkla výpočetní sílu GPU všeobecným výpočtům, toto nazýváme GPGPU (General Purpose GPU) a dnes se používá ve všech oblastech lidského rozvoje a výzkumu. Jako příklad můžeme uvést využití pro simulaci molekul v oblasti chemie či bioinformatiky, datová analýza a v neposlední řadě simulace a předpověď počasí. Ačkoli CUDA zpřístupnila GPU dalším odvětvím, neznamená to však, že by nepřispěla odvětví GPU vlastnímu, tedy počítačovým hrám. V kombinaci s PhysX, či přímým využitím, mohou vývojáři přesunout zátěž výpočtů fyziky, částic a dalších efektů z CPU na GPU a nabídnout tak hráčům mnohem realističtější prostředí plné interaktivních předmětů korektně reagujících na vlivy okolí.

6 Implementace

6.1 Úvod

Vzhledem k tomu, že stěžejní část programu je psaná v Cuda for C/C++, která plně nepodporuje třídy a vzhledem k rozsahu celého kódu nebyl kladen důraz na OOP. Převod do OOP a vytvoření rozhraní pro další aplikace, či grafické rozhraní by tak mohl být další krok k rozšíření tohoto programu.

6.2 Inicializace

V části programu vykonávané na CPU je implementována hlavní zobrazovací smyčka programu využívající otevřených knihoven GLEW a freeGLUT, které usnadňují práci s oknem a vykreslování scény, kterou vypočítal raytrace kernel na grafické kartě. Propojení rozhraní CUDA a OpenGL nám umožňuje rovnou vykreslovat data bez nutnosti je každý snímek přenášet z paměti GPU do paměti počítače. K tomu nám pomohou funkce z hlavičkového souboru cuda_gl_interop.h. Nejprve pomocí OpenGL vytvoříme vertex buffer object o velikosti plochy skrz kterou budeme vést paprsky při raytracingu a výsledný ukazatel předáme funkci cudaGLRegisterBufferObject(GLuint), která jej registruje na straně CUDy, abychom s ním mohli dále pracovat. Toto je pouze příprava, kterou je potřeba udělat v inicializační části programu před vstupem do hlavní smyčky. V hlavní smyčce poté pomocí cudaGLMapBufferObject(void**, GLuint) namapujeme registrovaný vertex buffer na ukazatel, který můžeme předat kernelu a ten do něj již může volně zapisovat. Po návratu z kernelu bychom měli buffer uvolnit analogickou funkcí cudaGLUnmapBufferObject(GLuint), která zajistí, že již do bufferu nebude ze strany CUDy zapisováno a může být tedy použit pro vykreslení obsahu bufferu pomocí OpenGL, kdy stačí data v bufferu reprezentovat jako souřadnice bodu a jeho barvu v RGBA. Nic kromě paměti nám samozřejmě nebrání alokovat větší buffer a ukládat do něj i další data, například pro postprocessing, ale v této implementaci si vystačíme se zmíněnými souřadnicemi a barvou. Touto metodou sdílení prostředků jsme minimalizovali množství dat přenášených každý snímek na zanedbatelný ukazatel a propojení CUDy a OpenGL nám usnadnilo práci s těmito daty.

6.3 Objekty

Objekty jsou reprezentovány jednotnou strukturou bez ohledu na typ, což znamená, že každý objekt zabírá zbytečný prostor, ale velikost objektů není ve stávající implementaci omezující, protože na grafickou kartu je přesunujeme pouze jednou a dnešní grafické karty disponují dostatečnou pamětí.

Jednotlivé objekty jsou při raytracingu rozlišeny podle přiřazeného typu a podle něj také využívají relevantní část dat. Do budoucna by bylo dobré vytvořit rozhraní, přez které bude raytracer k objektům přistupovat, což s omezeními platformy CUDA [12] znamená posílat na GPU čistá data a teprve zde vytvářet objekty.

6.4 Object loader

Dále je zde implementována jednoduchá třída s jedinou statickou metodou, jejíž účelem je nahrání souboru ve formátu obj do programu, umožňující snadno zobrazit scény připravené v programech určených pro práci s grafikou. Formát obj byl zvolen pro jeho jednoduchost a rozšířenost, protože je do něj schopna exportovat většina programů. Detailní popis podporovaného formátu a příklad nastavení exportu v programu Blender najdete v příloze C. Program prochází celý textový soubor a zahazuje nerozpoznané řádky, ve chvíli kdy narazí na řádek označující vertex, normálu či polygon, se jej pokusí zpracovat a přidat do seznamu pro další zpracování. Po průchodu celým souborem vytvoří ze zpracovaných dat seznam všech trojúhelníků s jejich normálami a vrátí jej volající funkci.

6.5 KD strom

6.5.1 Definice

KD strom je datová struktura v podobě binárního stromu organizující body v K-dimenzionálním prostoru. Každý uzel v tomto stromu představuje bod v prostoru, kterým je vedena rovina, jenž podle aktuální dimenze dělí body v prostoru a všechny body, které mají v této dimenzi nižší souřadnici než uzel, spadají do levého podstromu a všechny s vyšší souřadnicí spadají do pravého podstromu. Body se stejnou souřadnicí mohou v závislosti na účelu KD stromu spadat do určeného podstromu, či do obou. KD stromy nacházejí využití všude, kde je potřeba vyhledávání podle K-dimenzionálního klíče. Ray tracer jej využije pro efektivní procházení prostoru a radikální snížení počtu testovaných objektů při hledání průsečíků s paprskem. Ve výpisu 2 vidíme algoritmus vytváření KD stromu využívající medián jako pozici dělící roviny.

```
function kdtree ( list of points pointList , int depth)
{
    int        axis = depth % k;
    float median = getMedian(axis,pointList);
    tree_node node;
    node.location = median;
    node.leftChild = kdtree(points in pointList before median, depth+1);
    node.rightChild = kdtree(points in pointList after median, depth+1);
    return node;
}
```

Výpis 2: Algoritmus tvorby KD sromu

6.5.2 Implementace

Urychlovací struktura KD stromu se vytváří na CPU a vzhledem k tomu, že pracujeme pouze se statickou scénou, se hned v inicializaci přenáší na grafickou kartu spolu s informacemi o objektech ve scéně. Vlastní implementace stavby KD-stromu je rekurzivní, kdy vytvořením prvního uzlu se z něj stává kořen a ten si již sám vytvoří všechny další uzly

až k listům, rozdělující rovina je vybírána naivně jako prostřední objekt po jejich seřazení v daném rozměru. Pro každou scénu je pro optimální výsledek potřeba definovat maximální hloubku stromu a počet objektů v listu stromu, ten také závisí na relativní obtížnosti mezi porovnáním paprsku s uzlem stromu a objektem ve scéně. V každém kroku jsou tak objekty daného uzlu seřazeny podle aktuální dimenze, a poté podle porovnání s dělící rovinou uloženy do levého, pravého, či obou uzlů. Ten pokud nedosáhl maximální hloubky, nebo počet objektů v uzlu není menší než určený počet objektů v listu, pokračuje stejným způsobem dále, jinak se stává listem a vrací svůj výsledek rodiči. V dalším rozšíření programu by bylo možné zkvalitnit strom a urychlit tak vlastní raytracing použitím mediánu objektů a také se pokusit eliminovat co nejvíce prázdných prostorů ve scéně, či použít složitější algoritmus jako například SAH (Surface Area Heuristic) [7], který ovšem negativně ovlivní dobu stavby KD-stromu, což nás nemusí u statické scény příliš zajímat. V případě interaktivní scény bychom již museli zvážit zdali doba stavby stromu nepřekročí urychlení, které poskytuje ve fázi raytracingu.

6.5.3 Datová reprezentace

Jednotlivé uzly stromu obsahují údaje o ose, podle které rozdělují prostor dělící rovinou a souřadnici, kterou tato rovina prochází. Dále obsahuje odkazy na levý a pravý uzel a rodiče, který momentálně není využit, ale byl by, kdybychom chtěli implementovat průchod stromem bez využití haldy. Nakonec obsahuje pole s odkazy na objekty a informaci o jejich skutečném počtu, vzhledem k tomu, že pole je alokované staticky. Všechny odkazy jsou řešeny indexy v poli, kde se nachází všechny objekty, protože ukazatele z hlavní paměti by nám na grafické kartě nebyly k ničemu. Statická alokace pole s odkazy na objekty nám usnadňuje přesun dat na GPU.

6.6 RayTrace kernel

6.6.1 Start ray tracingu

V inicializační části se na grafické kartě nastaví ukazatelé na pole s objekty ve scéně, kořen KD stromu, pozice kamery a promítací plocha. Také zde definujeme světla ve scéně. Vlastní raytrace kernel poté spočítá příslušný paprsek podle pozice v mřížce vláken a započne testování průsečíku s objekty ve scéně. Nejdříve z důvodu optimalizace testuje průsečík s koulí obsahující všechny objekty ve scéně. Vzhledem k tomu, že jsme při tvorbě KD stromu nijak prázdný prostor neřešili, můžeme se tímto způsobem jednoduše zbavit velké části paprsků, které jdou mimo oblast, ve které se nachází objekty pokud nezabírají celou plochu zobrazované scény.

6.6.2 Průchod KD stromu

Po průchodu tímto úvodním testem již přichází na řadu samotný průchod KD stromem, který v implementaci vypadá složitě, ale v zásadě se jedná pouze o zachycení všech 7 možných případů, jak daný paprsek prochází současným uzlem. Počítáme s minimálními a maximálními rovinami, které vymezují jeho hranice, pro jednoduchost budou dále

v textu označovány jako čelní a zadní rovina, bráno z pohledu paprsku. U prvních třech uzlů jsou nastaveny na největší vzdálenost objektů od dělící roviny, hlouběji ve stromu je pak již určují předcházející dělící roviny. Možnosti které mohou při průchodu uzlem nastat:

- Paprsek protíná dělící rovinu před čelní rovinou, počátek tedy leží mimo uzel a prochází pouze vzdálenějším potomkem, tento případ je jednoduchý a pokračujeme do tohoto potomka.
- Paprsek protíná dělící rovinu za zadní rovinou a zároveň protíná zadní rovinu, počátek opět leží mimo uzel a tentokrát prochází pouze bližším uzlem, pokračujeme do tohoto potomka.
- Paprsek protíná dělící rovinu v záporném směru (čelní před dělící) a protíná zadní rovinu, v tomto případě může počátek ležet uvnitř uzlu, procházíme bližšího potomka.
- Paprsek protíná dělící a čelní rovinu v záporném směru (dělící před čelní) a protíná zadní rovinu, v tomto případě může počátek ležet uvnitř uzlu, procházíme bližšího potomka.
- Paprsek leží na dělící rovině, podle složky směru paprsku v dimenzi aktuálního uzlu procházíme bližšího (složka větší než nula) nebo vzdálenějšího (složka je menší než nula) potomka.
- Paprsek protíná dělící rovinu mezi čelní a zadní rovinou, v tomto případě procházíme bližší uzel a vzdálenější umístíme na haldu, protože bude následovat, pokud v bližším uzlu nenajdeme průsečík.

Když dojdeme do listu stromu čeká nás výpočet průsečíků v něm. V tomto procesu již lineárně procházíme seznam objektů a testujeme je na průsečík. Ve chvíli kdy najdeme průsečík, tak v závislosti na tom jestli hledáme nejbližší, nebo jakýkoliv (výpočet stínu) jej vrátíme z funkce, nebo pokračujeme v testování. Jak upozorňuje práce [6], tak po průchodu všech objektů v uzlu, ještě ale nemusíme mít nejbližší průsečík. Pokud leží průsečík na polygonu a ten patří i do vzdálenějšího uzlu, může průsečík na něm ležet až za dělící rovinou (vzdálenost průsečíku je větší, než vzdálenost dělící roviny rodiče) a v tom případě musíme otestovat i objekty ze vzdálenějšího uzlu, protože zde může být bližší průsečík. Pokud průsečík nenajdeme, tak se podíváme na haldu a pokud obsahuje nějaký uzel pokračujeme jeho průchodem. Pokud je halda prázdná, paprsek žádný objekt neprotíná a vracíme prázdný průsečík.

```
begin
   (entry distance, exit distance) <− intersect ray with root AABB;
   if ray does not intersect AABB then
      return no object intersected;
   end
   push (tree root node. entry distance, exit distance) to stack;
   while stack is not empty do
```

```
(current node, entry distance, exit distance) <- pop stack;
while current node is not a leaf do
    a <- current node split axis;
    t <- (current node split position.a - ray origin.a) / ray dir.a;
    (near,far) <- classify near/far with (split position.a > ray origin.a);
    if t>= exit distance or t < 0 then
        current node <- near;
    else if t <= entry distance then
        current node <- far;
    else
        push(far,t, exit distance) to stack;
        current node <- near;
        exit distance <- t;
    end
end
if current node is not empty leaf then
    intersect ray with each object;
    if any intersection exists inside the leaf then
    return closest ojbect to ray origin;
end
end
return no object intersected;
end
```

Výpis 3: Algoritmus průchodu stromem, Zdroj: [6]

6.6.3 Získání barvy vzorku

Pokud jsme našli průsečík projdeme všechny světla ve scéně a zjistíme jestli paprsek z místa průsečíku k nim protíná nějaký objekt dříve, něž dorazí ke světlu. Pokud průsečík nenajdeme, připočteme k barvě příspěvek tohoto světla pomocí phongova osvětlovacího modelu [10]. Dále k výsledné barvě připočteme barvu tělesa a ořežeme ji pokud by měla překročit rozsah. Poté ji ještě vynásobíme odrazivostí materiálu tělesa a dostáváme výsledný příspěvek k barvě. Nyní spočítáme odražený paprsek a rekurzivně pokračujeme v raytracingu. Ve chvíli kdy dosáhneme maximální hloubky rekurze, minimálnímu příspěvku k barvě v závislosti na odrazivosti, nebo již nenajdeme další průsečík, spočítáme výslednou barvu paprsku jako součet barev všech paprsků lomeno součtem odrazivostí a tuto barvu vracíme skrze všechny úrovně až do základní funkce, která je převede na RGB v rozsahu 0-255 a zapíše je do našeho pole vzorků.

6.7 Průsečík s polygonem

Pro získání průsečíku paprsku s polygonem nejdříve získáme průsečík paprsku s rovinou polygonu a převedeme tak problém hledání průsečíku v trojrozměrném prostoru do dvourozměrného.

Rovinu můžeme zapsat vzorcem jako

$$ax + by + cz = d \tag{1}$$

Kde koeficienty a,b a c tvoří normálu roviny n, můžeme ji tedy zapsat jako

$$nx = d \tag{2}$$

Nyní definujeme paprsek z bodu P ve směru D

$$nx = d \tag{3}$$
$$R(t) = P + tD \tag{4}$$

K nalezení průsečíku paprsku a roviny dosadíme parsek do rovnice roviny za $x = R(t)$

$$n * R(t) = d \tag{5}$$
$$n * (P + tD) = d \tag{6}$$
$$nP + ntD = d \tag{7}$$
$$t = (d - nP)/nD \tag{8}$$

Po získání průsečíku s rovinou porovnáme jeho pozici vzhledem k jednotlivým hranám polygonu a pokud u všech tří leží na vnitřní straně polygonu, můžeme říct, že paprsek protíná polygon v tomto bodě.

Pro určení zda leží bod na vnitřní straně hrany použijeme následující nerovnici

$$((V2 - V1)x(Q - V1)) * n \geq 0 \tag{9}$$

Kde $V1$ a $V2$ jsou vrcholy tvořící hranu trojúhelníku a Q je bod v rovině trojúhelníku získaný v předcházejícím kroku.

Jedná se o velmi prosté řešení a tak jsem pro další urychlení zvolil poněkud komplikovanější, ale rychlejší algoritmus, který využívá barycentrických koordinát [8] s optimalizovaným výpočtem uvedeným v [9]. Pro srovnání jsem obě řešení s pomocí barycentrických koordinát i jednoduché porovnávání ponechal zakomentované v kódu.

6.8 Průsečík s koulí

U koule je získání průsečíku výrazně jednodušší, neboť kouli můžeme popsat jednou jednoduchou rovnicí a dosadit do ní rovnici paprsku. Začneme tedy rovnicí pro výpočet všech bodů na povrchu koule:

$$(X - C)(X - C) = r2 \tag{10}$$

Kde X je bod na povrchu koule, C je střed a r je poloměr koule. Za X tedy dosadíme již zmíněnou rovnici paprsku 4

$$(P + tD - C)(P + tD - C) = r2 \tag{11}$$
$$t2(D * D) + 2t(D(P - C)) + (P - C) * (P - C) = r2 \tag{12}$$
$$t2(D * D) + 2t(D(P - C)) + (P - C) * (P - C) - r2 = 0 \tag{13}$$

Zde již vidíme, že se jedná o kvadratickou rovnici s koeficienty

$$a = (D * D) \tag{14}$$
$$b = (D * (P - C) \tag{15}$$
$$c = (P - C) * (P - C) \tag{16}$$

Kterou řešíme spočítáním diskriminantu d a následně pokud je nezáporný koeficientu t pro průsečíky s koulí.

$$t = \frac{-b \pm \sqrt{4ac - b^2})}{2a} \tag{17}$$

Pokud jsme směr paprsku D normalizovali můžeme za a rovnou dosadit 1 a vyjde nám tedy

$$t = \frac{-b \pm \sqrt{4c - b^2}}{2} \tag{18}$$

6.9 Antialaliasing

6.9.1 Aliasing

Aliasingem rozumíme rušivou chybu v signálu způsobenou jeho rekonstrkucí z omezeného vzorku [13]. V grafice jej můžeme vnímat jako „zubaté“ okraje, nebo tzv. Moirého vzory. V raytracingu k tomuto jevu dochází proto, že jedním nekonečně tenkým paprskem reprezentujeme barvu celého pixelu. Pro omezení těchto rušivých jevů používáme antialiasing.

6.10 Antialiasing

Antialiasing je souhrný název pro řadu metod, jak zabránit projevům aliasingu. V této implementaci budeme používat antialiasing prováděný pomocí supersamplingu, tedy metody, kdy na jeden výsledný pixel připadá větší množství vzorků. V rasterizaci je tato metoda řešena tím, že se scéna vykresluje do několikanásobně většího množství pixelů, které se poté průměrují na výsledný pixel na obrazovce. Podobně tomu bude i zde, protože pro každý výsledný pixel budeme vysílat větší množství paprsků.

6.10.1 Implementace

Abychom získali první vzorky pro antialiasing co nejsnáze, rozhodl jsem se vést paprsky rohy pixelů místo jejich středů a první úroveň antialiasingu tak můžeme provést bez počítání dalších vzorků. Pokud rozdíl barev mezi vzorky překročí danou hranici zjemníme mřížku a spočítáme pomocí raytracingu vzorky pro nově vzniklé pozice, ty opět ve čtveřicích porovnáme a pro kvadranty které mají stále příliš rozdílné barvy (hrany, ostré přechody) pokračujeme rekurzivně v antialiasingu dokud nejsme s jemností přechodu spokojeni, nebo nedosáhneme zvolené maximální úrovně. Poté co dorazíme na konec, ať už hloubkou nebo splněním kritérií, zprůměrujeme barvy ze čtyř vzorků a tuto barvu

vrátíme. Tímto dosáhneme vyhlazení ostrých hran a přechodů, aniž bychom museli počítat do maximální hloubky v místech, kde je přechod mezi barvami sousedních pixelů plynulý. Počet nově počítaných vzorků je minimalizován použitím vzorků z předchozí úrovně. Antialiasing by bylo možné v budoucnu dále optimalizovat sdílením okrajových vzorků mezi vlákny, ale to by již vyžadovalo pokročilé sdílení vypočítaných vzorků a složitější synchronizaci vláken.

Rozdíl mezi barvami počítám jako vzdálenost mezi jednotlivými barvami, což je pouze aproximací faktického rozdílu mezi barvami, tak jak jej vnímá lidské oko. V budoucnu by bylo pro zlepšení kvality výsledků dobré tento jednoduchý výpočet nahradit výpočtem ΔE, což je oficiální míra pro určení člověkem vnímaného rozdílu mezi dvěma barvami podle CIE (Mezinárodní komise pro osvětlování) [11]. Barvy ovšem musí být první převedeny z RGB do formátu XYZ a následně L*a*b*, z kterého teprve můžeme vypočítat ΔE, což přinese další zátěž pro samotný antialiasing.

7 Výsledky

V této kapitole uvádím souhrn všech testů a jejich výsledků které jsem na implementovaném raytraceru provedl.

7.1 Testovací sestava

- **GPU:** NVIDIA GeForce 660GTX 2GB GDDR5
- **CPU:** Intel i5-3570 3.4GHz
- **Paměť:** 16GB DDR3
- **Základní deska:** H77MA-G43
- **Operační systém:** Windows 7 Ultimate SP1 64bit

7.2 Scény

Pro testování jsem vybral tři scény, které testují engine v určitém směru.

7.2.1 Scéna č.1 - Defaultní scéna

Scéna je tvořena lehce pootočenou krychlí složenou ze 125 koulí (5x5x5). V této scéně je dobře viditelná hloubka rekurze a její adaptivní kontrola skrze jednotlivé úrovně odrazů na koulích, také jsou zde dobře viditelné stíny. Scéna má nízký počet objektů, které zabírají velký prostor a tak při průchodu KD stromem často testujeme jeden objekt vícekrát, protože koule zasahují do více sousedních listů stromu, což vysvětluje horší výkon při použití KD stromu v této scéně. Tato scéna se spustí, pokud uživatel nezadá programu vlastní soubor na vstup. Render této scény je vidět na obrázku 1.

7.2.2 Scéna č.2 - Opička Suzanne

V této scéně se nachází referenční model hlavy opičky Suzanne, který je často používán pro demonstrace a testy v počítačové grafice. Model obsahuje 968 polygonů a zastupuje tak optimalizovaný průměrně složitý model používaný v real-time počítačové grafice. Na této scéně je již znatelný přínos při použití KD stromu. Render této scény je vidět na obrázku 2.

7.2.3 Scéna č.3 - Stanfordský králík

Tato scéna je opravdovým zátěžovým testem raytraceru, nachází se v ní model Stanfordského králíka, který je též používám jako referenční model v počítačové grafice. Zde je použit ve verzi s 69630 polygony a při jeho použití je vidět, že stavba KD stromu zabírá mnohem více času, než samotný raytracing. Render této scény je vidět na obrázku 3.

7.3 Výsledná data

V této podkapitole uvádím výsledná data získaná při testování raytraceru na výše zmíněné sestavě ve všech uvedených scénách. Snažil jsem se data vybrat tak, aby byl vidět časový rozdíl pro danou scénu a testovaný prvek implementace.

7.3.1 Adaptivní hloubka

V tabulce 1 můžeme vidět výsledky testování vlivu adaptivní hloubky raytracingu na dobu vykreslování scény. K testování byla použita scéna č.1 s koulemi, která obsahuje dostatek objektů, mezi kterými se může paprsek odrážet. Z grafu náležícího k tabulce je patrné, že do určitého bodu je doba výpočtu s i bez použití adaptivní hloubky velmi podobná, avšak u varianty s adaptivní hloubkou začne nárůst doby vykreslování snímku klesat s tím, jak má stále méně paprsků dostatečnou intenzitu na to, aby byl počítán další odraz. Nakonec se ve scéně bez dokonale odrazivých objektů ustálí na určité hodnotě, protože žádný, nebo minimum paprsků pokračuje do větší hloubky.

Hloubka	0	1	2	3	4	5	6	7	8	9	10
Plná hloubka	22	33	41	49	55	66	68	75	83	84	91
Adaptivní hloubka	22	33	35	38	38	41	41	41	41	41	41

Tabulka 1: Adaptivní hloubka

Graf 4 k této tabulce můžete najít v příloze B této práce.

7.3.2 KD strom

Pro test efektivity KD stromu jsem zvolil scény č.2 a 3, protože disponují velkým počtem polygonů a tak by zde měl být vliv KD stromu jasně viditelný. Pro potřeby postupného zvyšování počtu polygonů ve scéně jsem postupně upravoval model opičky Suzanne ze scény č.2 pomocí nástroje Subdivision aplikovaného na její části v grafickém programu Blender. Obdobně jsem pomocí nástroje Decimate zjednodušoval původní model Stanfordského králíka. Testovací vzorky s 986 až 18674 polygony vycházejí z modelu opičky Suzanne a zbývající od 27070 do 69630 polygonů vychází z modelu Stanfordského králíka. Z tabulky 2 je vidět, že bez použití KD stromu je růst času vykreslování lineární (počet objektů, které jsme schopni za časový úsek zpracovat je konstantní), kdežto při použití KD stromu, jsme schopni s každou úrovní stromu zpracovat větší množství objektů jediným testem na průsečík s daným uzlem.

Počet polygonů	986	1713	2864	3872	4844	6926
Bez KD stromu	347	596	1020	1398	1751	2510
S KD stromem	84	121	136	150	160	203
Počet polygonů	10744	18674	27070	36094	50132	69630
Bez KD stromu	4380	6973	8746	11687	16259	22561
S KD stromem	242	328	442	522	621	782

Tabulka 2: KD Strom

Graf 5 k této tabulce můžete najít v příloze B této práce.

7.3.3 Adaptivní antialiasing

Jak bylo naznačeno již v části implementace, antialiasing samotný je náročný proces, protože s každou úrovní exponencionálně roste počet paprsků, které musíme dopočítat a jeho adaptivní varianta je v implementaci stále zatížena tím, že nesdílí data mezi jednotlivými vlákny a vzorky potřebné pro antialiasing daného pixelu jsou již počítány lineárně. Od CUDA 5.0 je dostupná dynamická paralelizace, kdy může kernel vyvolat nový více vláknový kernel, ale funkce je dostupná pouze na zařízeních s compute capability 3.5 a výše, což jsou karty řady Tesla a některé z nejvýkonnějších karet řady GeForce, ke kterým jsem neměl přístup. Adaptivní antialiasing je velmi závislý na množství ostrých přechodů a hran ve scéně, proto jsem se rozhodl jej otestovat na všech scénách vždy s nastavením, které umožňovalo nejrychlejší raytracing. Výsledky můžeme vidět v tabulkách 3, 4 a 5. V grafu 6 pak můžeme vidět průměrné zlepšení z těchto testů.

Hloubka	0	1	2	3	4	5
Plný AA	22	24	615	2978	12374	49945
Adaptivní AA	22	24	155	559	1739	4356

Tabulka 3: Adaptivní Antialiasing Scéna č.1

Hloubka	0	1	2	3	4
Plný AA	83	84	1463	6697	27956
Adaptivní AA	79	82	275	902	2730

Tabulka 4: Adaptivní Antialiasing Scéna č.2

Hloubka	0	1	2	3	4
Plný AA	444	444	7725	36678	152962
Adaptivní AA	444	444	1018	2717	6972

Tabulka 5: Adaptivní Antialiasing Scéna č.3

8 Závěr

I přez svou jednoduchou impementaci je KD strom nejvíce znatelným urychlením raytraceru, které je markantní zvláště u náročnějších scén s vysokým počtem objektů. V těchto scénách relativně náročnějším testováním uzlu stromu odstraňujeme takové množství objektů, jehož sekvenční testování by zabralo mnohonásobně víc času. Bez použití KD stromu by testovací scénu s opičkou Suzanne nebylo možné vykreslovat v reálném čase a Stanfordský králík by byl vykreslován několik desítek vteřin. Možností pro zlepšení je v oblasti KD stromu dostatek a pro výběr optimálního řešení je zapotřebí hlouběji prozkoumat metody pro efektivní konstrukci kvalitního KD stromu a jeho průchodu. Neopomíjel bych ani možnost nahrazení KD stromu jinou strukturou, jako BVH [14] nebo Octree [15].

Vliv adaptivní hloubky raytracingu je znát hlavně ve scénách s větším počtem matných objektů a vyšší maximální hloubkou rekurze, kdy je na výsledcích vidět urychlení předčasným ukončením paprsků dopadajících na tyto tělesa. V případě scény plné dokonale lesklých těles nemá adaptivní hloubka na čas vykreslování vliv.

Antialiasing samotný je velkou přítěží pro vykreslování scény a zpomalení vykreslování je značné a tak lze o urychlení mluvit pouze ve srovnání s jeho adaptivní verzí, která se snaží jednoduchým způsobem identifikovat aliasingem postižená místa a omezit tak množství upravovaných pixelů. Kvalitu antialiasingu by šlo v budoucnu zvýšit použitím přesnějšího postupu pro určení rozdílu mezi barvami [11]. Další optimalizace je v této oblasti určitě možná a to jak urychlením samotného raytracingu (KD strom) na kterém je antialiasing závislý, tak pokročilým sdílením dat, synchronizací mezi jednotlivými vlákny antialiasing kernelu a paralelním výpočtem dalších vzorků.

Implementace dosahuje reálných časů i na složitějších statických scénách, pokud vynecháme antialiasing. Pro reálné nasazení a dynamickou scénu je zatím nevhodná, ale po aplikaci navržených vylepšení a optimalizací by měla být schopna dosahovat uspokojivých výsledků i v této oblasti.

9 Reference

[1] Arthur Appel *Some techniques for shading machine renderings of solids*, New York: Yorktown Heights, 1968.

[2] Turner Whitted *An Improved Illumination Model for Shaded Display*, New Jersey: Holmdel 1979.

[3] POHL, Daniel. Quake Wars* Gets Ray Traced. POHL, Daniel. Quake Wars* Gets Ray Traced. *Intel® Developer Zone* [online]. Intel, 2.7.2012 [cit. 2014-04-30]. Dostupné z: https://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/

[4] GORAL, Cindy M., Kenneth E. TORRANCE, Donald P. GREENBERG a Bennett BATTAILE. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics*. 6.1984, roč. 18, č. 3. Dostupné z: http://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S07/lectures/goral.pdf

[5] LAFORTUNE, Eric. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. Katholieke Universiteit Leuven, 1996. Dostupné z: http://graphics.cs.kuleuven.be/publications/ERICLPHD/. Disertační práce. Katholieke Universiteit Leuven.

[6] HAPALA, M. a V. HAVRAN. Review: Kd-tree Traversal Algorithms for Ray Tracing. *Computer Graphics Forum*. 2011, roč. 30, č. 1. Dostupné z: http://dcgi.felk.cvut.cz/home/havran/ARTICLES/cgf2011.pdf

[7] WALD, Ingo a Vlastimil HAVRAN. *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)*. 2006. Dostupné z: http://www.eng.utah.edu/~cs6965/papers/kdtree.pdf

[8] CURLESS, Brian. *Ray-triangle Intersection*. 2006. Dostupné z: http://courses.cs.washington.edu/courses/cse457/07sp/lectures/triangle_intersection.pdf

[9] ERICSON, Christer. *Real-time collision detection*. Vyd. 1. Boston: Morgan Kaufmann, 2005, 591 s. ISBN 15-586-0732-3.

[10] WATT, Alan. *Advanced animation and rendering techniques: theory and practice*. Vyd. 1. New York: Addison-Wesley, 1992, 455 s. ISBN 02-015-4412-1.

[11] SHARMA, Gaurav. *Digital color imaging handbook*. Boca Raton, FL: CRC Press, c2003, 797 p. ISBN 08-493-0900-X.

[12] E.2.6.3. Virtual Functions. In: *Programming Guide :: CUDA Toolkit Documentation* [online]. 13.2.2014 [cit. 2014-05-04]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#virtual-functions

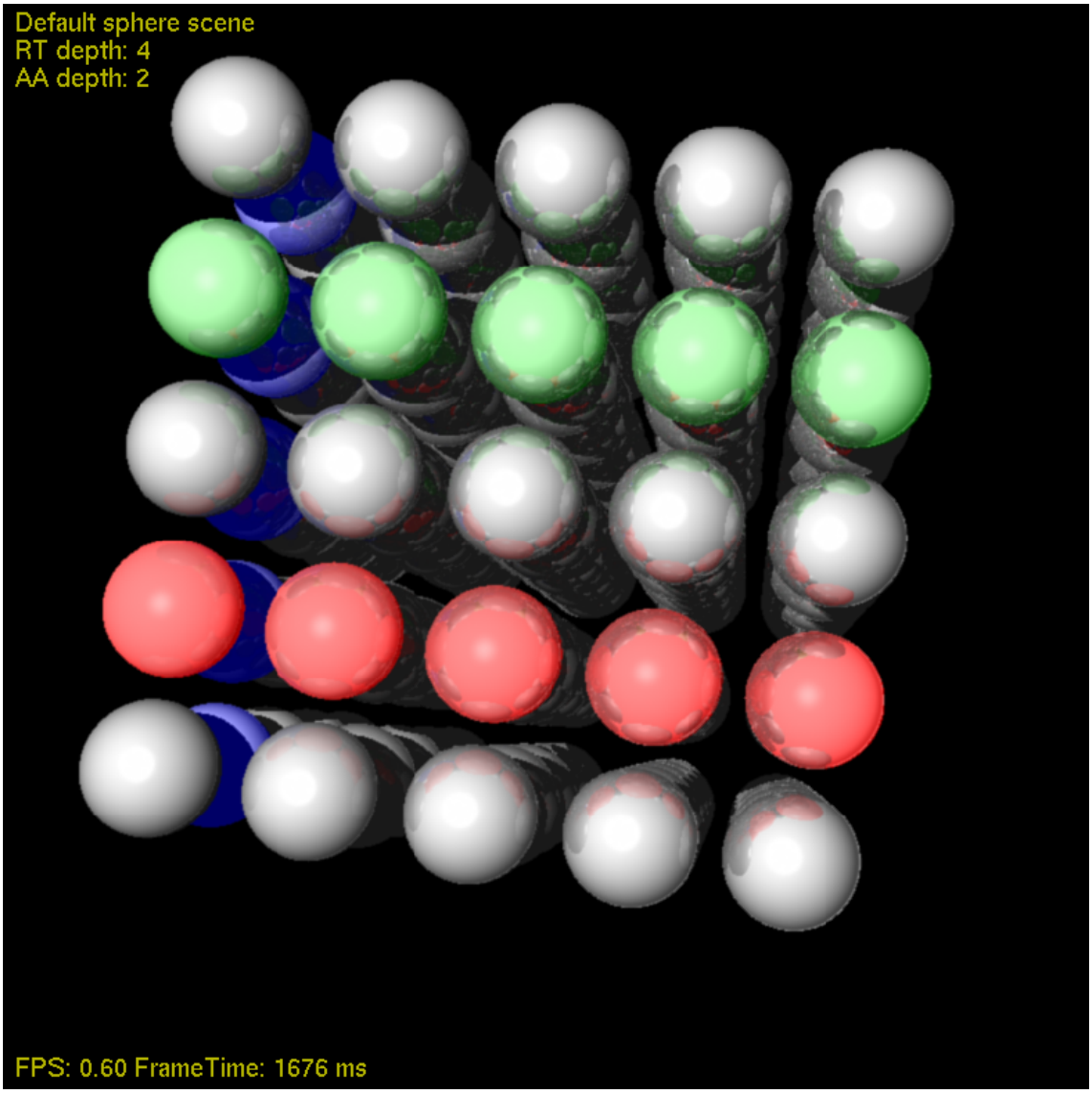
[13] MITCHELL, Don P. a Arun N. NETRAVALI. *Reconstruction Filters in Computer Graphics*. 1988 [cit. 2014-05-04]. Dostupné z: http://mentallandscape.com/Papers_siggraph88.pdf

[14] GÜNTHER, Johannes, Stefan POPOV, Hans-Peter SEIDEL a Philipp SLUSALLEK. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In: *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*. 2007. Dostupné z: http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/BVHonGPU.pdf

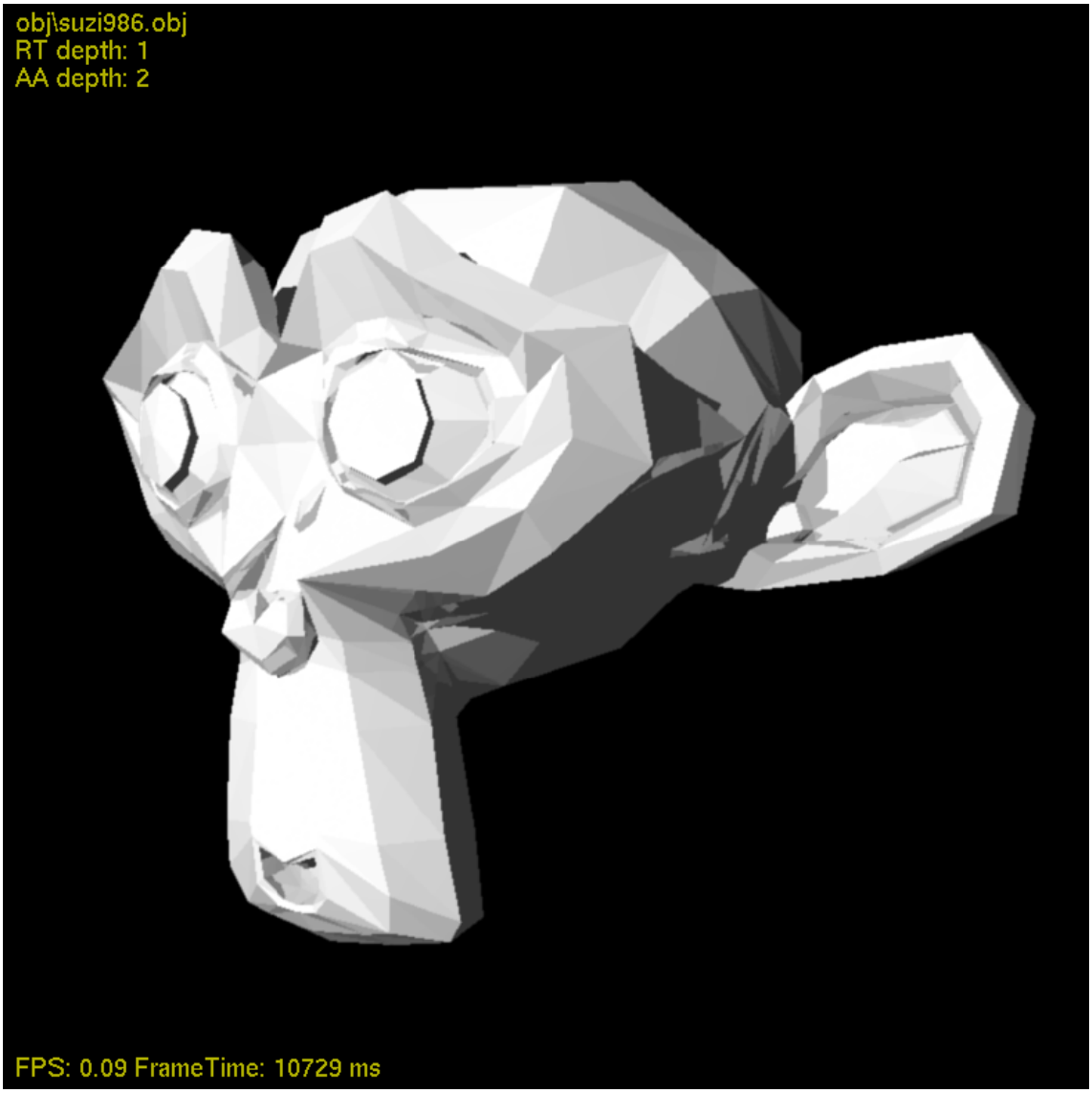
[15] MEAGHER, Donald. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980. Technical Report IPL-TR-80-111. Rensselaer Polytechnic Institute.

A Rendery scén

V této příloze se nachazí obrázky všech renderovaných scén odkazovaných v textu.



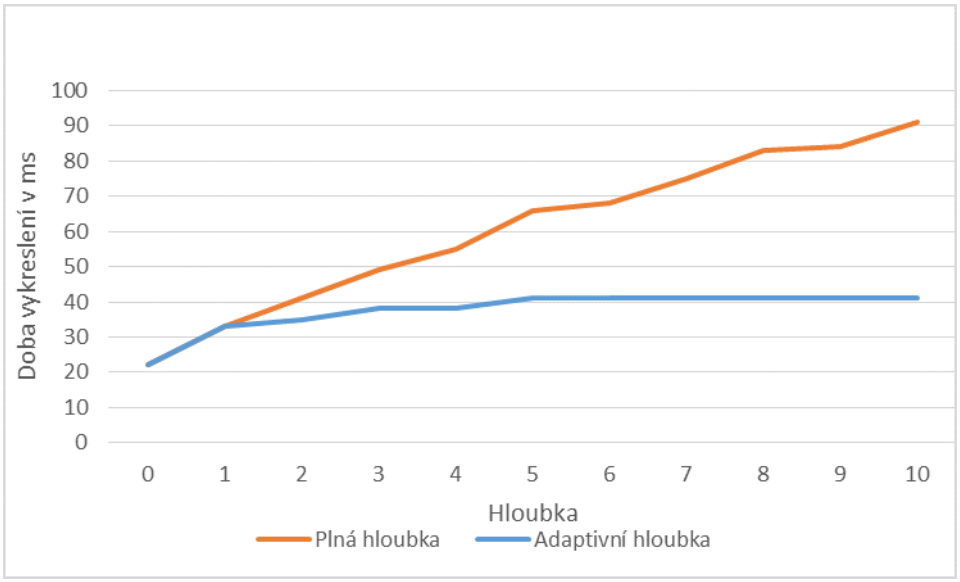
Obrázek 1: Scéna č.1 - Defaultní scéna



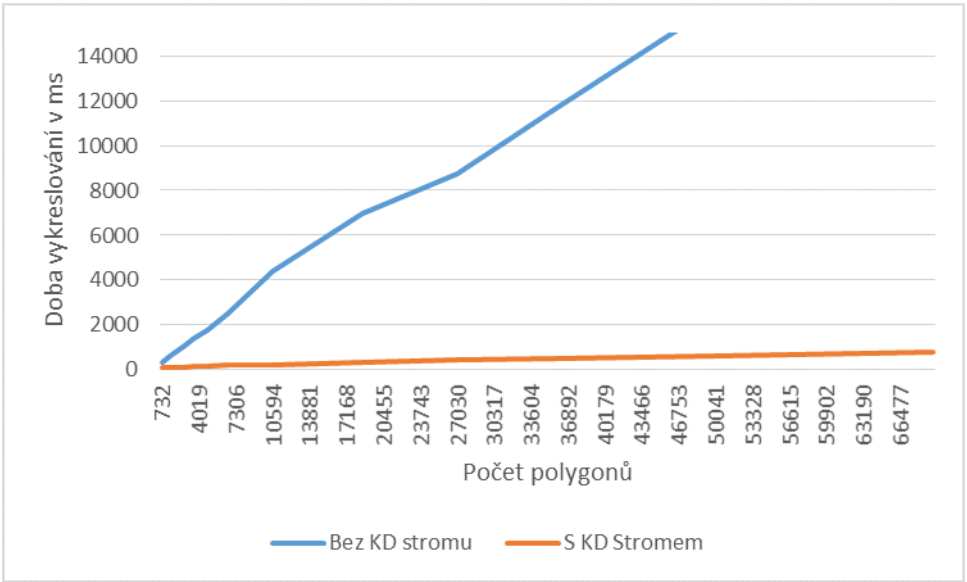
Obrázek 2: Scéna č.2 - Opička Suzanne

Obrázek 3: Scéna č.3 - Sandfordský králík

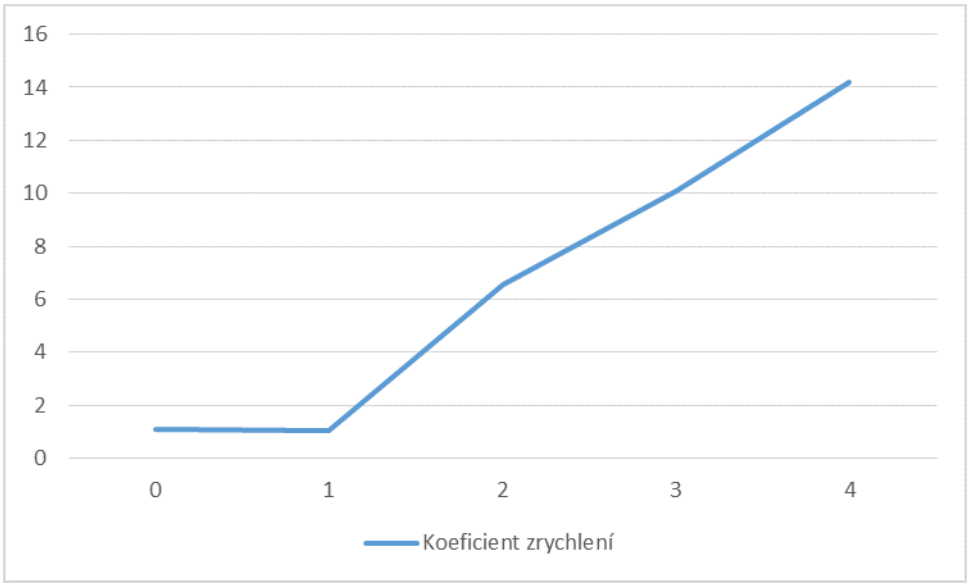
B Grafy



Obrázek 4: Adaptivní hloubka



Obrázek 5: KD Strom



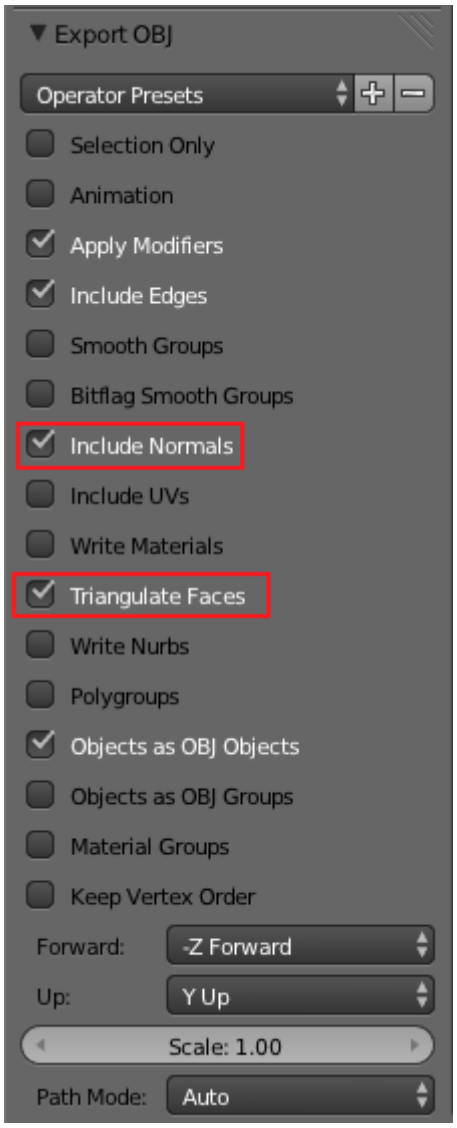
Obrázek 6: Adaptivní Antialiasing

C Uživatelská dokumentace

Běh programu lze ovlivňovat mnoha parametry, zde uvádím jejich výčet a popis.

- „-fd n“ -určuje vzdálenost průmětny od kamery, defaultní hodnota 1000
- „-vp n“ -velikost okna / 2 ,defaultní hodnota 400 (okno 800*800)
- „-rt n“ -hloubka rekurze raytracingu, defaultní hodnota 2
- „-aa n“ -hloubka rekurze antialiasingu, defaultní hodnota -1 (zcela vypnutý)
- „-obj file“ -cesta k obj souboru k nahrání (defaultně prázdný)
- „-adaptiveAA“ -zapne adaptivní antialiasing
- „-aDepth“ -zapne adaptivní hloubku
- „-tree“ -zapne použití KD stromu
- „-pos n n n“ -určuje pozici objektu nahraného ze souboru, defaultně 0,0,0

Na obrázku 7 můžeme vidět nastavení pro export obj souboru z programu Blender ve formátu, který dokáže program zpracovat.



Obrázek 7: Nastavení exportu v Blenderu

D Obsah přílohy CD

Obsah složek:

- „Objeky“ - objekty použité při testování.
- „Program“ - spustitelný program zkompilovaný pro Win32. Před spuštěním může být potřeba nainstalovat vcredist_x86.exe pokud na PC není Visual Studio.
- „Skripty“ - skripty použité pro získání dat pro kapitolu 7.
- „TextPrace“ - text této práce ve formátu PDF i zdrojový LATEX soubor.
- „ZdrojovyKod“ - zdrojové kódy s projektem pro Visual Studio 2012.
- „Dokumentace“ - programátorská dokumentace projektu.

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu